GRENOBLE 1
UNIVERSITE
JOSEPH FOURIER
SCIENCES. TECHNOLOGIE. SANTÉ

INF121:
Functional Algorithmic and Programming
Lecture 4: Recursion

Academic Year 2011 - 2012

In the previous episodes of INF 121

- Basic Types:

Type	Operations	Constants
Booleans	`not, &&, ||`	`true, false`
Integers	`+,-,*,/,mod, ...`	`...,-1, 0, 1, ...`
floats	`+.,-.,*.,/.`	`0.4, 12.3, 16. , 64.`
char	`lowercase, code, ...`	`'a', 'u', 'A', ...`

- `if` ... `then` ... `else` ... conditional structure
- identifiers (local and global)
- defining and using functions
- Advanced types: synonym, enumerated, product, sum
- Pattern matching on simple and advanced expressions

About recursion

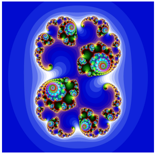
What is recursion/a recursive definition?

About recursion

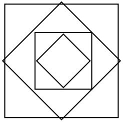
What is recursion/a recursive definition?

Example (Some recursive objects)

La vache qui rit is a trademark



$u_n, u_{n+1} \ldots$
Fibonacci



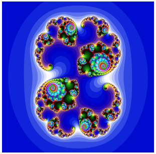
Images under Creative Common License

About recursion

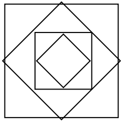
What is recursion/a recursive definition?

Example (Some recursive objects)

La vache qui rit is a trademark



$u_n, u_{n+1} \ldots$
Fibonacci



Images under Creative Common License

Recursive functions generalize recursive series

Largely used in Computer Science
$\hookrightarrow$ a computer is a zoo of interacting recursive functions

Outline

- Recursive functions
- Termination
- Recursive types
- Conclusion

Recursive functions in OCaml

An introductory example

Example (Factorial)

$$\begin{cases} 0! = 1 \\ n! = n \times (n-1)!, n \geq 1 \end{cases}$$

$$\begin{aligned} 3! &= 3 \times (3-1)! = 3 \times 2! \\ &= 3 \times 2 \times (2-1)! = 3 \times 2 \times (2-1)! \\ &= 3 \times 2 \times 1 \times (1-1)! = 3 \times 2 \times 1 \times 0! = \ldots = 6 \end{aligned}$$

- This definition is sensible, it allows to obtain a result for all integers: *well-founded*
  (changing the $-$ into $+$ in the 2^{nd} line makes the def not well-founded)

Recursive functions in OCaml

An introductory example

Example (Factorial)

$$\begin{cases} 0! = 1 \\ n! = n \times (n-1)!, n \geq 1 \end{cases}$$

$$\begin{aligned} 3! \quad &= 3 \times (3-1)! = 3 \times 2! \\ &= 3 \times 2 \times (2-1)! = 3 \times 2 \times (2-1)! \\ &= 3 \times 2 \times 1 \times (1-1)! = 3 \times 2 \times 1 \times 0! = \ldots = 6 \end{aligned}$$

- This definition is sensible, it allows to obtain a result for all integers: *well-founded*
  (changing the $-$ into $+$ in the 2^{nd} line makes the def not well-founded)

How can we detect whether a function or a program is well-founded?

Recursive functions in OCaml

An introductory example

Example (Factorial)

$$\begin{cases} 0! = 1 \\ n! = n \times (n-1)!, n \geq 1 \end{cases}$$

$$\begin{aligned} 3! &= 3 \times (3-1)! = 3 \times 2! \\ &= 3 \times 2 \times (2-1)! = 3 \times 2 \times (2-1)! \\ &= 3 \times 2 \times 1 \times (1-1)! = 3 \times 2 \times 1 \times 0! = \ldots = 6 \end{aligned}$$

- This definition is sensible, it allows to obtain a result for all integers: *well-founded*
  (changing the − into + in the 2nd line makes the def not well-founded)

How can we detect whether a function or a program is well-founded?

Example (Defining factorial in OCaml)

```
let rec fact (n:int):int =
  if n=0 then 1
  else n ∗ fact(n−1)
```

```
let rec fact (n:int):int =
  match n with
    0 → 1
    | n → n ∗ fact(n−1)
```

Defining a recursive function

Specification: description, signature, examples, and recursive equations

Implementation: defining a recursive function in OCaml

```
let rec fct_name (p1:t1) (p2:t2) ... (pn:tn):t = expr
```

where `expr` generally contains one or more occurrences of `fct_name` s.t.:

- Basis case: no call to the function currently defined
- Recursive calls to the currently defined function (with different parameters)

Defining a recursive function

Specification: description, signature, examples, and recursive equations

Implementation: defining a recursive function in OCaml

```
let rec fct_name (p1:t1) (p2:t2) ... (pn:tn):t = expr
```

where `expr` generally contains one or more occurrences of `fct_name` s.t.:

- Basis case: no call to the function currently defined
- Recursive calls to the currently defined function (with different parameters)

Typing works as for non-recursive functions

Remark

- `t1`, .., `tn` can be any type (not necessarily integers) - cf. later
- A recursive function *cannot* be anonymous

□

Defining some recursive functions

Example (Sum of integers from 0 to n)

description + profile + examples

$$\begin{cases} u_0 & = 0 \\ u_n & = n + u_{n-1} \quad \text{when } 0 < n \end{cases}$$

```
let rec sum (n : int) : int =
 match n with
  | 0 → 0
  | n → n + sum (n − 1)
```

Defining some recursive functions

Example (Sum of integers from 0 to n)

description + profile + examples

$$\begin{cases} u_0 & = 0 \\ u_n & = n + u_{n-1} \quad \text{when } 0 < n \end{cases}$$

```
let rec sum (n : int) : int =
 match n with
  | 0 → 0
  | n → n + sum (n − 1)
```

Example (Quotient of the Euclidian division)

description + profile + examples

$$a/b = \begin{cases} 0 & \text{when } a < b \\ 1 + (a - b)/b & \text{when } b \leq a \end{cases}$$

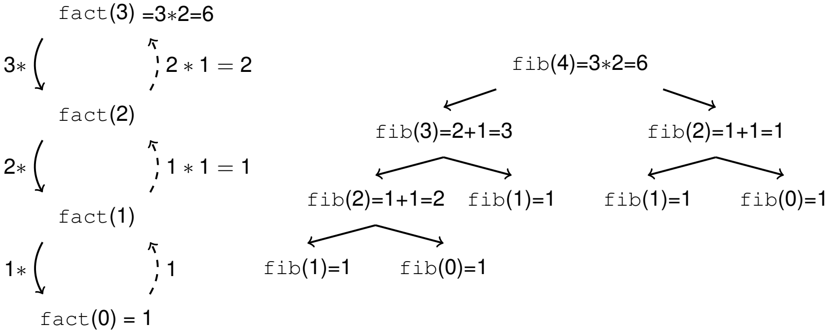
```
let rec div (a : int) (b: int) : int =
 if a < b then 0
  else 1 + div (a − b) (b)
```

DEMO: some other recursive functions

Calling a recursive function

"Unfolding the function body" – rewriting

Example (Factorial and Fibonacci's call trees)



`fact`(3) =3∗2=6

3∗ … $2 * 1 = 2$

`fact`(2)

2∗ … $1 * 1 = 1$

`fact`(1)

1∗ … 1

`fact`(0) = 1

`fib`(4)=3∗2=6

`fib`(3)=2+1=3

`fib`(2)=1+1=1

`fib`(2)=1+1=2

`fib`(1)=1

`fib`(1)=1

`fib`(0)=1

`fib`(1)=1

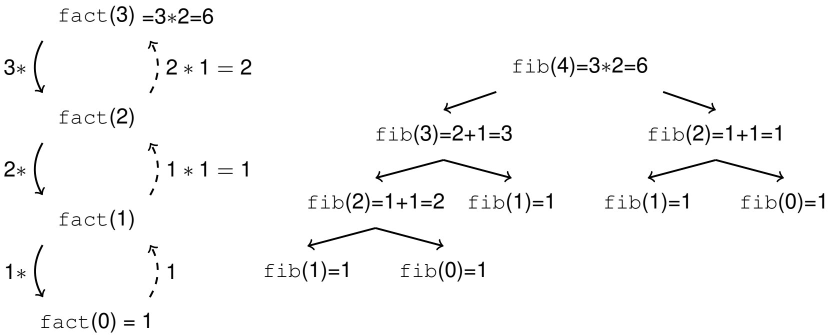
`fib`(0)=1

- $\longrightarrow$: rewriting generated calls and suspending operations
- $\dashrightarrow$: evaluation (in the reverse order) of suspended operations

Calling a recursive function

"Unfolding the function body" – rewriting

Example (Factorial and Fibonacci's call trees)



`fact`(3) =3∗2=6

3∗ $2 * 1 = 2$

`fact`(2)

2∗ $1 * 1 = 1$

`fact`(1)

1∗ 1

`fact`(0) = 1

`fib`(4)=3∗2=6

`fib`(3)=2+1=3

`fib`(2)=1+1=1

`fib`(2)=1+1=2

`fib`(1)=1

`fib`(1)=1

`fib`(0)=1

`fib`(1)=1

`fib`(0)=1

- $\longrightarrow$: rewriting generated calls and suspending operations
- $\dashrightarrow$: evaluation (in the reverse order) of suspended operations

In OCaml: directive #`trace`

DEMO: Tracing a function

Let's practice

Exercise: remainder of the Euclidean division

Define a function which computes the remainder of the Euclidean division

Exercise: The Fibonacci series

Implement a function which returns the n^{th} Fibonacci number where n is given as a parameter. Formally the Fibonacci series is defined as follows:

$$fib_n = \begin{cases} 1 & \text{when } n = 0 \text{ or } n = 1 \\ fib_{n-1} + fib_{n-2} & \text{when } n > 1 \end{cases}$$

Let’s practice

Exercise: the power function (two ways)

$$\begin{cases} x^0 & = 1 \\ x^n & = x * x^{n-1} \quad \text{when } 0 < n \end{cases} \qquad \begin{cases} x^0 & = 1 \\ x^n & = (x * x)^{n/2} \quad \text{when } n \text{ is even} \\ x^n & = x * (x * x)^{\frac{n-1}{2}} \quad \text{when } n \text{ is odd} \end{cases}$$

- Define function `power`: `int` → `int` → `int` twice following the two equivalent mathematical definitions
- What is the difference between those two versions?

Let's practice

Exercise: the power function (two ways)

$$\begin{cases} x^0 & = 1 \\ x^n & = x * x^{n-1} \quad \text{when } 0 < n \end{cases}$$

$$\begin{cases} x^0 & = 1 \\ x^n & = (x * x)^{n/2} \quad \text{when } n \text{ is even} \\ x^n & = x * (x * x)^{\frac{n-1}{2}} \quad \text{when } n \text{ is odd} \end{cases}$$

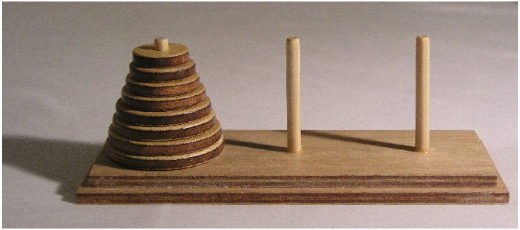
- Define function `power`: `int` → `int` → `int` twice following the two equivalent mathematical definitions
- What is the difference between those two versions?

```
let rec pow (x:float) (n:int):int =
  if (n=0) then 1
  else x * (pow x n−1)
```

```
let rec pow (x:float) (n:int):int =
 if (n=0) then 1
 else (
  if (n mod 2=0) then (pow (x*x) (n/2))
  else pow (x*x) ((n−1)/2)
 )
```

The Hanoi towers

A word about Divide and Conquer



Images under Creative Common License

Mutually recursive functions

On an example

So far “direct” recursion: a function `fct` contains calls to itself

What about a function `f` which calls `g` which calls `f`
↪ mutually recursive functions

Mutually recursive functions

On an example

So far "direct" recursion: a function `fct` contains calls to itself

What about a function `f` which calls `g` which calls `f`
↪ mutually recursive functions

Example (Is a number odd or even)

How to determine whether an integer is odd or even without using
/, $*$, `mod`,and, more specifically using $-$ and =?

Mutually recursive functions

On an example

So far “direct” recursion: a function `fct` contains calls to itself

What about a function `f` which calls `g` which calls `f`
↪ mutually recursive functions

Example (Is a number odd or even)

How to determine whether an integer is odd or even without using /, $*$, `mod`,and, more specifically using $-$ and =?

- $n \in \mathbb{N}$ is odd if $n - 1$ is even
- $n \in \mathbb{N}$ is even if $n - 1$ is odd
- 0 is even
- 0 is not odd

Mutually recursive functions

On an example

So far “direct” recursion: a function `fct` contains calls to itself

What about a function `f` which calls `g` which calls `f`
$\hookrightarrow$ mutually recursive functions

Example (Is a number odd or even)

How to determine whether an integer is odd or even without using /, $*$, `mod`,and, more specifically using $-$ and =?

- $n \in \mathbb{N}$ is odd if $n-1$ is even
- $n \in \mathbb{N}$ is even if $n-1$ is odd
- 0 is even
- 0 is not odd

```
let rec even (n:int):bool = if n=0 then true else odd (n−1)
  and odd (m:int):bool = if m=0 then false else even (m−1)
```

DEMO: even and odd, mutually recursive

Mutually recursive functions

Generalization

Mutually recursive functions

```
let rec fct1 [parameters+return type] = expr_1
  and fct2 [parameters+return type] = expr_2
     ....
  and fctn [parameters+return type] = expr_n
```

`expr_1`, `expr_2`, ..., `expr_n` may have calls to `fct1`, `fct2`, ..., `fctn`

Outline

Recursive functions

Termination

Recursive types

Conclusion

Termination

Do you think this function terminates (the McCarthy function)?

$$mac(n) = \begin{cases} n - 10 & \text{when } n > 100 \\ mac(mac(n + 11)) & \text{when } n \leq 100 \end{cases}$$

Termination

Do you think this function terminates (the McCarthy function)?

$$mac(n) = \begin{cases} n - 10 & \text{when } n > 100 \\ mac(mac(n + 11)) & \text{when } n \leq 100 \end{cases}$$

What about these ones?

The power function

$$\begin{cases} x^0 & = 1 \\ x^n & = x * x^{n-1} \quad \text{when } 0 < n \end{cases}$$

The factorial function

$$\begin{cases} fact(0) & 1 \\ fact(1) & = 1 \\ fact(n) & = \frac{fact(n+1)}{n+1} \end{cases}$$

Termination

Do you think this function terminates (the McCarthy function)?

$$mac(n) = \begin{cases} n - 10 & \text{when } n > 100 \\ mac(mac(n + 11)) & \text{when } n \leq 100 \end{cases}$$

What about these ones?

The power function

$$\begin{cases} x^0 & = 1 \\ x^n & = x * x^{n-1} \quad \text{when } 0 < n \end{cases}$$

The factorial function

$$\begin{cases} fact(0) & 1 \\ fact(1) & = 1 \\ fact(n) & = \frac{fact(n+1)}{n+1} \end{cases}$$

We are only interested in terminating functions...

Can we have an intuitive characterization of termination w.r.t. the calling tree?

How can we prove that a recursive function terminate?

Using a Measurement

Theorem

Every series of positive numbers which is <u>strictly</u> decreasing is converging

How can we prove that a recursive function terminate?

Using a Measurement

Theorem

Every series of positive numbers which is strictly decreasing is converging

General methodology to show a function is terminating

From the def. of the function and its parameters, derive a measurement s.t.:

- it is positive
- the measurement *strictly decreases* between two recursive calls

each recursive call “brings us closer to the base case”

How can we prove that a recursive function terminate?

Using a Measurement

Theorem

Every series of positive numbers which is strictly decreasing is converging

General methodology to show a function is terminating

From the def. of the function and its parameters, derive a measurement s.t.:

- it is positive
- the measurement *strictly decreases* between two recursive calls

each recursive call "brings us closer to the base case"

Example (Termination of the function `sum`)

```
let rec sum (n : int) : int =
  match n with
   | 0 → 0
   | n → n + sum (n − 1)
```

Measurement:

- Let's define $\mathcal{M}(n) = n$
- $\mathcal{M}(n) \in \mathbb{N}$ (according to the spec)
- $\mathcal{M}(n) > \mathcal{M}(n-1)$ since $n > n-1$

Termination of some functions

Exercise: finding measurements

Revisit the functions `factorial`, `power`, `quotient`, `remainder` and find the measurement proving that your function terminates

Termination of some functions

factorial and power

Termination of `fact`:

```
let rec fact (n:int):int =
  match n with
    0 → 1
    | n → n ∗ fact(n−1)
```

Termination of some functions

factorial and power

Termination of `fact`:

```
let rec fact (n:int):int =
  match n with
    0 → 1
    | n → n ∗ fact(n−1)
```

- Let's define $\mathcal{M}(\textit{fact}\ n) = n$
- $\mathcal{M}(\textit{fact}\ n) \in \mathbb{N}$ (according to the spec)
- $\mathcal{M}(\textit{fact}\ n) > \mathcal{M}(\textit{fact}\ (n-1))$ since $n > n-1$

Termination of some functions

factorial and power

Termination of `fact`:

```
let rec fact (n:int):int =
  match n with
    0 → 1
    | n → n * fact(n−1)
```

- Let's define $\mathcal{M}(\mathit{fact}\ n) = n$
- $\mathcal{M}(\mathit{fact}\ n) \in \mathbb{N}$ (according to the spec)
- $\mathcal{M}(\mathit{fact}\ n) > \mathcal{M}(\mathit{fact}\ (n-1))$ since $n > n-1$

Termination of `power`:

```
let rec power (a:float) (n:int):float =
  if (n=0) then 1.
 else (if n>0 then a *. power  a (n−1)
    else 1. /. (power a (n−1))
  )
```

Termination of some functions

factorial and power

Termination of `fact`:

```
let rec fact (n:int):int =
  match n with
    0 → 1
    | n → n ∗ fact(n−1)
```

- Let's define $\mathcal{M}(\mathit{fact}\ n) = n$
- $\mathcal{M}(\mathit{fact}\ n) \in \mathbb{N}$ (according to the spec)
- $\mathcal{M}(\mathit{fact}\ n) > \mathcal{M}(\mathit{fact}\ (n-1))$ since $n > n-1$

Termination of `power`:

```
let rec power (a:float) (n:int):float =
  if (n=0) then 1.
 else (if n>0 then a ∗. power  a (n−1)
    else 1. /. (power a (n−1))
  )
```

- Let's define $\mathcal{M}(\mathit{power}\ a\ n) = n$
- $\mathcal{M}(\mathit{power}\ a\ n) \in \mathbb{N}$ (according to the spec)
- $\mathcal{M}(\mathit{power}\ a\ n) > \mathcal{M}(\mathit{power}\ a\ (n-1))$

Termination of some functions

```
let rec quotient (a:int) (b:int):int =
 if (a<b) then 0
 else 1 + quotient (a−b) b
```

```
let rec remainder (a:int) (b:int):int =
  if (a<b) then a
  else remainder (a−b) b
```

Termination of some functions

```
let rec quotient (a:int) (b:int):int =
 if (a<b) then 0
 else 1 + quotient (a−b) b
```

```
let rec remainder (a:int) (b:int):int =
  if (a<b) then a
  else remainder (a−b) b
```

Termination of `quotient` and `remainder`:

- Let's define $\mathcal{M}(X\,a\,b) = a$
- $\mathcal{M}(X\,a\,b) \in \mathbb{N}$ (according to the spec)
- $\mathcal{M}(X\,a\,b) > \mathcal{M}(X\,(a-b)\,b)$ since $b > 0$

where $X \in \{\text{quotient}, \text{remainder}\}$

Outline

- Recursive functions
- Termination
- Recursive types
- Conclusion

Recursive types

Recursive functions are functions that appear in their own definition

Recursive types are types that appear in their own definition

Recursive types

Recursive functions are functions that appear in their own definition

Recursive types are types that appear in their own definition

General syntax: `type new_type` = ... `new_type`...

Recursive types

Recursive functions are functions that appear in their own definition

Recursive types are types that appear in their own definition

General syntax: `type new_type` = ... `new_type`...

Recursive types should be well-founded

They make sense only for Union type with a non recursive constructor (constant or not)

DEMO: (not) Well-founded types

DEMO: Metaphor of building a wall

Recursive types

Recursive functions are functions that appear in their own definition

Recursive types are types that appear in their own definition

General syntax: `type new_type` = ... `new_type`...

Recursive types should be well-founded

They make sense only for Union type with a non recursive constructor (constant or not)

DEMO: (not) Well-founded types

DEMO: Metaphor of building a wall

Definition of a recursive function on a recursive type should follow the recursive type

A recursive type: Peano natural numbers

The mathematical and OCaml perspectives

Peano natural numbers $\mathrm{NatPeano}$: an alternative definition of $\mathbb{N}$

Recursive definition of $\mathrm{NatPeano}$:

- a basis natural Zero
- a *constructor*: Suc: returns the successor of a $\mathrm{NatPeano}$ number
- Zero is the successor of no $\mathrm{NatPeano}$ number
- two $\mathrm{NatPeano}$ numbers having the same successor are equal

$\hookrightarrow$ $\mathbb{N}$ can be defined as the set containing Zero and the successor of any element it contains

A recursive type: Peano natural numbers

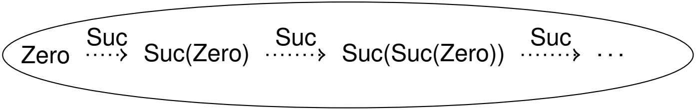
The mathematical and OCaml perspectives

Peano natural numbers $\mathrm{NatPeano}$: an alternative definition of $\mathbb{N}$

Recursive definition of $\mathrm{NatPeano}$:

- a basis natural Zero
- a *constructor*: Suc: returns the successor of a $\mathrm{NatPeano}$ number
- Zero is the successor of no $\mathrm{NatPeano}$ number
- two $\mathrm{NatPeano}$ numbers having the same successor are equal

$\hookrightarrow$ $\mathbb{N}$ can be defined as the set containing Zero and the successor of any element it contains



$$\begin{tikzcd} \text{Zero} \arrow[r, dotted, "\text{Suc}"] & \text{Suc(Zero)} \arrow[r, dotted, "\text{Suc}"] & \text{Suc(Suc(Zero))} \arrow[r, dotted, "\text{Suc}"] & \cdots \end{tikzcd}$$

Defining $\mathrm{NatPeano}$ in OCaml:

```
type natPeano = Zero | Suc of natPeano
```

$\hookrightarrow$ `natPeano` is a **recursive sum type**

Peano natural numbers

Conversion to and from integers

Example (Converting a Peano natural number into an integer)

- Description: `natPeano2int` translates a Peano number into its usual counterpart in the set of integers
- Profile/Signature: `natPeano2int`: `natPeano` → `int`
- Ex.: `natPeano2int Zero` = 0, `natPeano2int Suc(Suc(Suc Zero))`=3

```
let rec natPeano2int (n:natPeano):int =
 match n with
    Zero → 0
   | Suc (nprime) → 1+ natPeano2int nprime
```

Peano natural numbers

Conversion to and from integers

Example (Converting a Peano natural number into an integer)

- Description: `natPeano2int` translates a Peano number into its usual counterpart in the set of integers
- Profile/Signature: `natPeano2int: natPeano → int`
- Ex.: `natPeano2int Zero` = 0, `natPeano2int Suc(Suc(Suc Zero))`=3

```
let rec natPeano2int (n:natPeano):int =
 match n with
    Zero → 0
   | Suc (nprime) → 1+ natPeano2int nprime
```

Example (Converting an integer into a Peano number)

Same as above but in the converse sense:

```
let rec int2natPeano (n:int):natPeano=
  match n with
   0 → Zero
   | nprime → Suc (int2natPeano (n−1))
```

DEMO: Functions natPeano2int and int2natPeano

Peano natural numbers

Some functions: sum, product

Exercise: sum of two Peano numbers

- Define the function that *sums two Peano numbers* without using the conversion from/to int
- Prove that your function terminates

Exercise: product of two Peano numbers

- Define the function that *multiplies two Peano numbers*
- Prove that your function terminates

Exercise: factorial of a Peano number

- Define the function that computes the *factorial* of a Peano number
- Prove that your function terminates

A recursive type: polynomials of 1 variable

A polynomial of one variable (a sum of monomials):

$$\alpha_n X^n + \alpha_{n-1} X^{n-1} + \ldots + \alpha_1 X^1 + \alpha_0$$

A recursive type: polynomials of 1 variable

A polynomial of one variable (a sum of monomials):

$$\alpha_n X^n + \alpha_{n-1} X^{n-1} + \ldots + \alpha_1 X^1 + \alpha_0$$

Let's see it as a recursive object: a polynomial is either a monomial or the sum of monomial and another polynomial

Model 1:

```
type coef = int
type degree = int
type monomial = coef * degree
type polynomial = Mn of monomial
          | Plus of monomial * polynomial
```

DEMO: Model 1 of Polynomials + its disadvantages

A recursive type: polynomials of 1 variable - ctd

Model 2:

- with canonical representation
- no monomial with null coefficient

```
type polynomial = Zero | Plus of monomial * polynomial

let well_formed (p:polynomial):bool = ...
(* checks order of coef + no null coeff *)
```

A recursive type: polynomials of 1 variable - ctd

Model 2:

- with canonical representation
- no monomial with null coefficient

```
type polynomial = Zero | Plus of monomial * polynomial
```

```
let well_formed (p:polynomial):bool = ...
(* checks order of coef + no null coeff *)
```

Exercise: Some functions around polynomials

- Define a function that checks whether a polynomial is well-formed, by:
  - checking that there is no null coefficient
  - degrees are given in decreasing order
- Degree max: Propose a new implementation of the function degree max supposing that a polynomial is well-formed
- Addition of two polynomials:
  - Define a function that performs the addition between a polynomial and a monomial
  - Define a function that performs the addition between two polynomials

Conclusion

Recusion: a fundamental notion

There are two forms of recursion in computer science:

- recursive functions
    - recursive equations
    - termination
    - definition = spec (description, profile, recursive equations, examples) + implem + terminations
    - pitfalls
- Recursive types/values/objects
    - definition
- Recursive functions on recursive types